KES.U.87.6

1

Kestrel Institute

Knowledge-Based Software Development Tools

by

Douglas R. Smith

and

Thomas T. Pressburger

September 1986

Appeared in *Software Engineering Environments*,
P. Brereton, Ed., Ellis Horwood Ltd., Chichester, 1988, pp. 79-103.

This research was sponsored in part by the Office of Naval Research under Contract N00014-84-C-0473 and in part by the Defense Advanced Research Projects Agency, under contract N00014-81-C-0582 monitored by the Office of Naval Research. The views and conclusions contained in this paper are those of the author and should not be interpreted as representing the official policies, either expressed or implied of Kestrel Institute, DARPA, ONR, or the U.S. Government.

KESTREL INSTITUTE • 1801 PAGE MILL ROAD • PALO ALTO, CA 94304 • (415) 493-6871

Appeared in *Software Engineering Environments*,
P. Brereton, Ed., Ellis Horwood Ltd., Chichester, 1988, pp. 79–103.

Knowledge-Based Software Development Tools

Douglas R. Smith
Thomas T. Pressburger

Kestrel Institute
1801 Page Mill Road
Palo Alto, California 94304-1216

September 30, 1987

ABSTRACT

We describe some of the experimental knowledge-based software development tools under development at Kestrel Institute. In particular, systems for automatically performing algorithm design, deductive inference, finite differencing, and data structure selection are discussed. A detailed case study is presented that shows how these systems could cooperate in supporting the transformation of a formal specification into efficient code. The example treated is a schedule optimization problem.

1 Introduction

The purpose of a programming environment is to enhance programming productivity. A comprehensive environment provides integrated tools that assist with all aspects of the software development process: specification acquisition and development, implementation, testing, integration, maintenance, project management, and communication among the designers and users of the system. The Knowledge-Based Software Assistant (KBSA) approach [13] advocated the building of a comprehensive, integrated, software development environment that contained formalized knowledge about each aspect of the programming process. In particular, it advocated the following design principles.

1. The software development process should revolve around the development of formal, very-high-level specifications from which efficient code is synthesized. The specifications should be executable in order to allow testing. The maintenance process should concentrate on evolving the specifications and resynthesizing efficient implementations, rather than evolving the code.

2. All the objects relevant to the programming process, e.g. specifications at all levels of refinement, derivations, test cases and results, project plans, bug reports, should be captured in a knowledge-base.

3. There should be knowledge-based support for all facets of the programming process, including automated implementation. Here, the term *automated* means that the process is carried out by the machine, either automatically or under human guidance.

Focusing the software process on the development of very-high-level specifications factors the implementation of the problem from its specification. This should ameliorate the maintenance problem, because it is easier to maintain perspicuous, modular specifications that lack the implementation detail of source code.

Automatic compilers for the specification language allow the designer to rapidly construct executable prototypes of a specification. Experience gained with such prototypes assures the designer that the specification is valid; i.e. that it specifies the desired behavior in the context in which the system is embedded.

Kestrel research has focused on (1) the architecture of such an environment, and (2) development of formal theories that can be used to support aspects of the software development process. The architecture of an environment embodying the KBSA design principles should be highly integrated; one where knowledge is explicit and manipulable by the environment itself. The architecture of a *self-described* programming environment was outlined in [27] (cf. [21]). We propose that a KBSA should contain the following components.

1. A *knowledge base manager* that manages representations of all software process objects, e.g. programs, including a representation of their abstract syntax annotated with dataflow and other assertions, derivation histories, mathematical facts, program transformation rules, bug reports, project schedules.

2. A single ***wide-spectrum, very-high-level language*** used to (1) express programs and specifications at all levels of refinement; (2) query and modify the environment's knowledge base; (3) express programming knowledge, such as program transformations, efficiency characteristics, and other facts. Such a language should integrate constructs from predicate calculus, set theory, and standard imperative programming languages.

3. A *formal body of knowledge* about the software development process. In particular, we propose that the implementation process be formalized as one of program refinement by the application of a body of program transformation rules. Implementation knowledge that is specific to an application domain is necessary also (e.g. [3,39]). Other software development aspects that could be formalized are performance estimation (e.g. [16]), version control (e.g. [28,26]), designer communication (e.g. [17]), task scheduling and project management (e.g. [25,6]).

The CHI system [36] developed at Kestrel was an approximation to this architecture. It contained a frame-oriented knowledge base manager, a partly-compilable language called **V** with a variety of constructs, and some theories that were codified in the language. In particular, encoded in CHI were theories for: generating data structure implementations [9], which will be mentioned again later; synthesizing database updates [14]; and automating the communication among designers [17]. The **V** language contained a transformation construct that was used to express program transformations.

The REFINE[1] system [1] is a commercially available environment, based on the architectural ideas in the CHI system. It contains an entity-attribute style database manager, accessed via the REFINE language. We have developed research prototypes using the REFINE environment that demonstrate algorithm design [35], logic compilation [38], program optimization [5], performance estimation [10], and project management, task scheduling, bug tracking, and version control [15].

This paper concentrates on specification implementation knowledge. We show how to derive an efficient implementation of a very-high-level specification of a scheduling problem. We show the use of the specification language, and how various kinds of implementation knowledge come into play in the derivation. Before the actual derivation is presented, two pieces of implementation knowledge will be described: the algorithm design tactic for *subspace generators* [33] as applied to optimization problems, and the *finite-differencing* program optimization technique. Implementation knowledge for data structure generation and performance estimation will be mentioned briefly.

Underlying our approach is the goal of an automatic programming system that needs little human guidance. This has been exemplified by our work in algorithm design. We have tried to explicate the structure of various classes of algorithms such as branch-and-bound and divide-and-conquer. Systems we have built can construct automatically (for the most part) these sorts of algorithms for a given problem specification. One feature of our approach is that our algorithm design tactics motivate the inferences and decision-making that have to be made to synthesize an algorithm. In contrast, work such as [8,23,37] is more concerned with correct, human-developed proofs and program derivations. In our approach, certain kinds of intermediate results (lemmas, equivalent expressions, lower bounds, etc.) are sought because general

[1]REFINE is a trademark of Reasoning Systems, Inc., Palo Alto, California.

knowledge about the type of algorithm being constructed allows them to be formally characterized. An inference system can then take the formal characterization and produce the desired result. Inference is only performed in highly constrained contexts and towards specific goals. This allows the tactics to work automatically. See [7,11,24] for surveys of approaches to specification implementation.

The specification/programming language used in this paper is loosely based on the REFINE^{TM} language. Specifications in this paper should be comprehensible because of the similarity of the language to standard programming and mathematical notation. A few notes are in order. The language contains the standard datatypes **integer**, **natno** (natural number) and the following datatype constructors.

seq(τ)	sequences of elements of type τ
set(τ)	sets of elements of type τ
bag(τ)	multisets of elements of type τ
binrel(τ_1, τ_2)	binary relations on $\tau_1 \times \tau_2$
$\tau_1 \rightarrow \tau_2$	mappings from τ_1 to τ_2

Examples of the sequence operations *concat*, *append*, *prepend*, and **reduce** follow: $concat([1,2],[3,4]) = [1,2,3,4]$; $append([1,2],3) = [1,2,3]$; $prepend([1,2],0) = [0,1,2]$; **reduce**$(+,[1,2,3]) = 1+2+3 = 6$. The expression *elements-of*(S) returns the elements in the sequence S as a bag. The standard operations on sets are provided, e.g. $\in$, $\cup$, $\{f(x) \mid P(x)\}$. Mappings may be defined using λ-expressions. The function **minimum**(f, S) returns some element of the set S that minimizes the function f. The form **assert** P **in** E announces that the assertion P holds upon entry to the expression E. Such an assertion suggests that E may be simplified because it occurs in a context where P holds.

This paper suggests, but cannot prove, the contention that such an environment will improve productivity. Our experience with the REFINE language and environment, and the experience of others with other very-high-level language-based environments, such as SETL [20], lend credence to such a contention.

2 Knowledge about Specification Implementation

We now introduce some techniques for designing algorithms and optimizing programs. These techniques are formal and automatable and are embodied in several implemented systems at Kestrel. The techniques require deductive inference only in a carefully controlled way called *directed inference*, which is also described.

2.1 Search and Optimization Algorithm Design Strategies

Many problems may be posed as *search* problems: given some input parameters $\vec{x}$ over an input domain D (i.e. $\vec{x} \in D$ holds), find an object z that satisfies $O(\vec{x}, z)$, where O is a given predicate called the *goal constraint*. Each such z satisfying the goal constraint is called a *feasible solution* to the search problem. The *generate-and-test* algorithm strategy is the most general kind of algorithm for solving such a search problem. It assumes that the sought-after object lies in a domain R, called the *solution space*, that can be enumerated. The so-called "British-Museum" generate-and-test algorithm simply enumerates each item z in R, returning a z for which $O(\vec{x}, z)$.

The goal of an *optimization* problem is to find a solution that is feasible, but, moreover, *optimal*, in some sense, over all the feasible solutions. The generate-and-test algorithm scheme can be applied to solve an optimization problem also: one enumerates all the feasible solutions searching for an optimal one.

A refinement of the British Museum algorithm strategy, called *subspace generato*, searches more efficiently by exploiting characteristics of the goal constraint O and the solution space R. To motivate this strategy, consider the problem of searching for a particular Early Greek statue in the British museum (given that we have lost our museum guide). The search can be made more efficient by exploiting the structure of the museum, that is, as a set of galleries. Information about a gallery may help to eliminate each object in the gallery as a possibility. For example, every object in a gallery labeled "Coins and Medals" can be eliminated a priori.

More formally, the subspace generator algorithm scheme requires that each of the following program items be synthesized.

1. a function *Initial-Subspace*$(\vec{x})$ that returns the initial solution space to be searched;
2. a function *Split*$(\vec{x}, \hat{r})$ that, given a subspace $\hat{r}$, returns a set $\{\hat{r}_1, \ldots, \hat{r}_n\}$ of smaller, disjoint subspaces;
3. an efficient necessary condition $\Phi(\vec{x}, \hat{r})$ that will serve to filter out those subspaces $\hat{r}$ that cannot contain a feasible solution;
4. a function for extracting an answer from a subspace. For example, if we have determined that the statue must be in gallery 409, then if we know that gallery 409 has only one object in it, we can easily close in on the statue in the gallery.

Once synthesized, these program items may be inserted into slots in an algorithm template, producing a program that provably solves the original problem.

We briefly show how a familiar problem can be solved by applying the subspace generator strategy. The problem is: given an element x and an ordered sequence S, find the index i at which x is located in S, i.e. $S(i) = x$. The initial solution space for this problem are the indices in the interval $[1 .. size(S)]$.

Each algorithm design strategies requires a library of supplementary methods. The subspace generator scheme requires methods for splitting a subspace of some particular datatype into subspaces. Several splitting functions are available for the data type of intervals. One way of subdividing an interval into subintervals is *singleton-split*: the interval $[l .. u]$ (where $u - l > 1$) is divided into the intervals $[l]$ and $[(l+1) .. u]$. Another is *split-in-half*: the interval $[l .. u]$ is split into $[l .. \lfloor \frac{l+u}{2} \rfloor]$ and $[\lfloor \frac{l+u+2}{2} \rfloor .. u]$. Because the given sequence S is ordered, there is an efficient necessary condition, described in the next section, for deciding whether a given subinterval $[l .. u]$ could possibly contain an index i for which $S(i) = x$. Once the splitting process divides the solution space into an interval of length 1, we can test whether the sole member of the interval is the desired index. The splitting function, necessary condition, etc. can be inserted into the subspace generator algorithm template to produce a search algorithm.

One critical item in synthesizing the subspace generator is the synthesis of the efficient necessary condition Φ where

$$\forall\, \vec{x}, \hat{r}\ [\vec{x} \in D \Longrightarrow (\exists\, z\ [z \in \hat{r} \wedge O(\vec{x}, z)] \implies \Phi(\vec{x}, \hat{r}))]. \qquad (1)$$

The justification for this condition is that if Φ fails on some subspace descriptor $\hat{r}$, i.e. $\neg\Phi(\vec{x}, \hat{r})$ holds, then by the contrapositive of (1) there are no feasible solutions in $\hat{r}$ and hence the entire subspace $\hat{r}$ can be safely eliminated from further consideration.

Branch-and-bound algorithms that find optimal solutions, not just feasible ones, can be constructed using the subspace generator scheme. A subspace $\hat{r}$ can be eliminated if it can be shown that every solution in the subspace has cost worse than that achieved by a solution that has already been found. The necessary condition is thus of the form $lb(\hat{r}) < ub$, where ub is the cost of the best solution found so far, and lb is a function that can compute, given a subspace $\hat{r}$, a lower bound on the cost of solutions in $\hat{r}$.

Necessary conditions and lower bound functions can be synthesized by applications of *directed inference*, described next.

2.2 Directed Inference

Directed inference generalizes the standard notions of theorem-proving and formula simplification. Given some assumptions A over variables $\vec{x}$, and a source formula F, the goal is to find a formula ϕ that (1) satisfies some syntactic constraints, e.g. its free variables $\vec{y}$ are a subset of a given set; (2) optimizes some cost function, e.g. in terms of syntactic simplicity and semantic strength[2]; and (3) that bears a specified

[2]A predicate $p(x)$ is *stronger* than $q(x)$—conversely, $q(x)$ is *weaker* than $p(x)$—if p is more restrictive than q, i.e. $\forall x\ [p(x) \Longrightarrow q(x)]$.

logical relationship e.g. $\Longrightarrow$, $\Longleftarrow$, $\Longleftrightarrow$, to F. For example, we might want to reason forward ($\Longrightarrow$) from F to find a consequent ϕ whose free variables are a subset $\vec{y}$ of $\vec{x}$ satisfying

$$\forall\, \vec{x}\, [A(\vec{x}) \Longrightarrow (F(\vec{x}) \Longrightarrow \phi(\vec{y}))]. \tag{2}$$

In other words, we try to find a necessary condition ϕ on F under assumptions $A(\vec{x})$.

Similarly, given $A(\vec{x})$ and a source expression $e(\vec{x})$, we might want to find another expression $\theta(\vec{y})$ that satisfies some syntactic constraints and satisfies some comparison relation, e.g. $\leq, \geq, =, \subseteq$, with e. For example, a lower bound θ for e over variables $\vec{y}$ satisfies:

$$\forall \vec{x}\, [A(\vec{x}) \Longrightarrow e(\vec{x}) \geq \theta(\vec{y})]. \tag{3}$$

In the context of directed inference, the logical relationship ($\Longrightarrow, \ldots$) or the comparison relation ($\leq, \ldots$) specifies the *direction* of the inference. For any particular direction, there is a tension between syntactic simplicity and semantic strength of the desired formula. For example, **true** is the simplest necessary condition for a given F, but is too weak; similarly, F itself is as strong as possible, but it isn't any simpler than F. We have developed an inference engine called RAINBOW [32] that finds formulas and expressions optimizing a heuristic measure of simplicity and strength. For algorithm design purposes, one seeks an expression that optimizes execution efficiency and semantic strength. Fortunately, syntactic simplicity and execution efficiency are correlated in the examples that follow.

In general we specify an inference as follows:

Assumptions	$A_1, A_2, \ldots, A_n$
Source	S
Inference-direction	$\longrightarrow$
Target-constraints	*varset* $\subseteq \{\ldots\} \wedge \ldots$
Cost-function	syntactic simplicity + ...

The target-constraints express conditions on the syntactic form of the desirable inferred term or formula T. The keyword *varset* denotes the set of free variables in T which we may want to restrict to a certain subset.

The problem of finding the efficient Φ in (1) can be specified as follows.

Assumptions	$A(\vec{x})$
Source	$\exists\, z\, [z \in \hat{r} \wedge O(\vec{x}, z)]$
Inference-direction	$\Longrightarrow$
Target-constraints	*varset* $\subseteq \{\vec{x}, \hat{r}\}$
Cost-function	syntactic simplicity + semantic strength

For example, in the array search problem above, the goal is to find an efficient test $\Phi(x, l, u, S)$ satisfying:

Assumptions	$sorted(S)$
Source	$\exists\, i\, [i \in [l\,..\,u] \wedge S(i) = x]$
Inference-direction	$\Longrightarrow$
Target-constraints	$varset \subseteq \{x, l, u, S\}$
Cost-function	syntactic simplicity + semantic strength

Our system derives the formula $S(l) \leq x \leq S(u)$. Incorporating this test and *split-in-half* into the subspace generator template yields the binary search algorithm. This algorithm requires time $O(\log(size(S)))$, whereas a naïve generate-and-test algorithm requires time $O(size(S))$.

2.3 Program Optimization and Finite Differencing

An expensive computation $E(\vec{x})$ on dependent variables $\vec{x}$ can be simplified by taking advantage of *context*. Context here means the invariant assertions that can be assumed to hold at some point in a program, and the values that have been computed and saved. In these cases, the directed inference system can be instructed to find a more efficient, specialized expression that is equal to $E(\vec{x})$, given the context assumptions. The role of context in optimization, and specialization transformations that make use of context, have been studied in [29].

For example, if the expression E contains a subexpression that has already been (or could be) computed and bound to a variable, then a common-subexpression elimination transformation allows the variable to be used in place of the subexpression in E. More formally, the expression $f(\vec{y})$ in $H(E_1(f(\vec{y}), \vec{x}_1), E_2(f(\vec{y}), \vec{x}_2))$ can be lambda abstracted, resulting in **let** $z = f(\vec{y})$ **in** $H(E_1(z, \vec{x}_1), E_2(z, \vec{x}_2))$.

A similar optimization opportunity arises if y is only an incremental modification of x, and $E(x)$ has already been calculated. In this case, it is often possible to calculate $E(y)$ given x, y, and $E(x)$ more efficiently than it is to calculate $E(y)$ just given y. An instance of this opportunity that is exploited in conventional compilers is the strength reduction optimization. One example of this optimization transforms a multiplication by a constant e.g. $E(y) = cy$, where $y = x + 1$ and $E(x)$ has already been calculated, to an addition of the constant, i.e. $E(y) = E(x) + c$. Another example, this time on the sequence datatype, transforms the calculation $size(y)$, where $y = append(x, a)$, for some element a, and where $size(x)$ has already been calculated, into the calculation $size(y) = size(x) + 1$. Notice the latter transformation has changed a linear time computation to one of constant time. In general, for an incremental update $U(x)$, we try to transform $E(U(x))$ into the computation $E_U(x, E(x))$, for some efficient E_U.

Function abstraction isolates the body of a function $f(x_1, \ldots, x_n)$ from the context set up in the functions that call it. In a purely applicative setting, the context that an expression $E(x_1, \ldots, x_n)$ is exposed to within a function f is determined solely by the bindings of f's parameters and the bindings of local variables. However, there is a transformation that can expose E to the context of the functions that call f. This transformation adds a new parameter, say c, to f's parameter list (now $f(x_1, \ldots, x_n, c)$). The new parameter c will be maintained equal to E. Any function g that contains a call to f, say the call is $f(U_1, \ldots, U_n)$, must now supply $E(x_1, \ldots, x_n)$: i.e., the call must be changed to $g(U_1, \ldots, U_n, E(U_1, \ldots, U_n))$. We also assert within f that $c = E(x_1, \ldots, x_n)$. Now that the expression E has been exposed to the context set up in g, it may be possible to simplify it. This transformation could be expressed as a composition of unfold, abstract, simplify, and fold steps as in [29].

A particularly important case of this transformation is when f calls itself, and $U_1, \ldots, U_n$, are incremental modifications to $x_1, \ldots, x_n$. In this case, application of the transformation results in exposing the E computation to a context containing the value of the E computation on similar values. Such *finite differencing* optimizations [22] can achieve significant space and time savings when they occur within loops or recursion, because some efficiency is gained each time f is called. The finite differencing optimization relies on algebraic properties of E and the incremental modifications. We use directed inference to express (in the single variable case) $E(U(x))$ equivalently as $E_U(x, E(x))$. The finite-differencing program transformation scheme used in this paper is illustrated informally below. It shows how the definitions for functions f and g would be changed.

```
g(y)=
  ...
  f(V)
  ...

f(x)=

  ...
  E(x)
  ...
  f(U(x))
  ...
```

$\longrightarrow$

```
g(y)=
  ...
  f(V, E(V))
  ...

f(x, c)=
 assert c = E(x) in
   ...
   c
   ...
   f(U(x), E_U(x, c))
   ...
```

Similar finite-differencing transformations expressed using second-order patterns have been given in [2].

The MEQ system at Kestrel [5] performs finite differencing transformations in an imperative setting. The optimization scheme we employ in this paper simplifies expensive expressions within recursive functions, as described above. We are working to develop tools that automatically locate the expensive, incrementally computable expressions. At present, the programmer indicates the expressions to be maintained.

Often, the expression E_h that incrementally computes the next value has itself some expensive operations that must be themselves maintained. Parameters holding these intermediate computations are also added to the recursive function and maintained.

Some expressions do not at first appear to be incrementally computable, but can be rewritten as equivalent, incrementally computable expressions [22]. For example, a standard technique for incrementally maintaining the expression $\forall x\ [P(x)]$ is to reformulate it equivalently as **size**$(\{x \mid \neg P(x)\}) = 0$. That is, the expression that is **true** if every x satisfies $P(x)$ is reformulated as the expression that is **true** only when the number of elements for which $\neg P(x)$ is zero. The number of elements x for which $\neg P(x)$ can more often be incrementally maintained than the original quantified expression.

After the program has been transformed to maintain intermediate expressions in parameters, it often turns out that some of the dependencies of expressions on others have been removed. Thus, some of the variables may be unused, or *dead*, hence they may be dropped from the parameter lists of the function. The variables that are dead can be detected by data flow analysis [22].

2.4 Data Structures

Another step in the algorithm design process is the development of representations for each of the abstract data types in the algorithm. Compilers typically provide a standard implementation representation for each type in their programming language. However as the level of the language rises, and higher-level data types, such as sets, sequences, and mappings, are included in the language, or as users specify their own abstract data types, standard representations cease being satisfactory. The difficulty is that the higher-level datatypes can be implemented in many different ways: e.g. sets may be implemented as lists, arrays, trees, etc. Depending on the mix of operations, their relative frequency of invocation, and size information, one implementation may be much better than another. Thus no single default implementation will give good performance for all specifications containing the abstract type. Work on data structure generation and selection for very-high-level languages attempts to deal with these problems [31,4].

Kotik's DSS system [9,19] can generate for a given very-high-level data type a large space of possible alternative representations expressed in terms of standard high level datatypes; e.g. integers, arrays, records. These alternative representations are generated by composing together small, well known datatype conversions (cf. [18]). For example, a set of elements may be represented as a sequence of those elements (repeated or not), or a set may be represented as a characteristic mapping; a mapping may be represented as a set of pairs, or as the composition of two other mappings; a sequence may be represented as a mapping from indices to the elements, or as a list. The DSS system was able to construct a representation such as *hash table whose buckets are lists* for the *set* datatype, just by composing these well-known conversions. Once some representation has been chosen, the program can be transformed into one using the standard datatypes.

Recent work [10] has extended DSS rules so that their applicability conditions take into account performance considerations. The representation decisions depend on the operations performed on it and their frequency of occurrence. For example, the main sequence operations in the searching program above are assigning the sequence, and accessing a random element—the latter being repeated many times. This suggests an array implementation, because random access takes constant time. [30] shows how a sequence may be implemented as a stack, and hence as an array with a pointer to the top, if only certain of the sequence operations are used, and subsequences are not referenced. This applies to our derivation, as discussed in Section 3.6. The subject of copy elimination, e.g. how to decide whether parameters need not be copied when passed, or whether certain actions can be performed in-place, has been discussed in [12]. We are refining a theory that guides the DSS system, but do not have a comprehensive theory as yet.

3 Example: Scheduling Jobs with Deadlines on a Single Processor

We illustrate the concepts and systems discussed above by stepping through the derivation of an optimization algorithm. All of the steps in this derivation are correctness-preserving so that the final code is guaranteed to correctly solve the problem described by the initial specification. Furthermore most of the programming knowledge used in the derivation is understood well enough that the steps are formally motivated. We believe that this derivation could be produced with very little guidance from a human developer. Most of the derivation steps can be performed by research systems currently running at Kestrel Institute.

3.1 Specification

Suppose that we wish to schedule a set of jobs on a processor subject to some constraints on the order in which jobs can run. Further suppose that each job completes in unit time, that each job has a deadline, and that we wish to minimize the number of jobs that fail to complete before their deadlines. If we define a *schedule* to be an ordering of a given set of jobs that satisfies some given constraints, then this is an optimization problem where the feasible space is the set of schedules, and the cost function is the number of jobs in a schedule that fail to complete before their deadline.

Formally, the feasible space of schedules can be specified as follows.

$$\begin{aligned}&\text{SCHEDULES}(Jobs : \mathbf{set}(JOB), Precedes : \mathbf{binrel}(JOB, JOB)) : \mathbf{set}(\mathbf{seq}(JOB)) = \\ &\quad \{S \mid S \in \mathbf{seq}(Jobs) \wedge \textit{elements-of}(S) = Jobs \wedge \\ &\qquad \forall j_1, j_2 \in Jobs\,[\,Precedes(j_1, j_2) \implies index(S, j_1) \leq index(S, j_2)]\}\end{aligned}$$

Here *Jobs* is the set of jobs that we wish to schedule. The variable *Precedes* is a binary relation over *Jobs* and is assumed to be a partial order. The feasible space SCHEDULES(*Jobs*, *Precedes*) is defined to be the set of all sequences S of *Jobs* whose elements (a multiset) are exactly *Jobs* and such that the ordering of jobs in S is consistent with the *Precedes* relation. Following is a specification of the optimal scheduling problem, called SWD (Scheduling With Deadlines).

$$\begin{aligned}&\text{SWD}(Jobs : \mathbf{set}(JOB), Precedes : \mathbf{binrel}(JOB, JOB), \\ &\quad Deadline : JOB \rightarrow \mathbf{natno}) : \mathbf{seq}(JOB) = \\ &\quad \mathbf{minimum}(C, \text{SCHEDULES}(Jobs, Precedes))\end{aligned}$$

The cost function C is

$$C(Jobs, Precedes, Deadline, S) = size(\{j \mid j \in Jobs \wedge Deadline(j) < index(S, j)\}).$$

The input *Deadline* is a mapping from jobs to deadline times (represented as natural numbers).

3.2 Development Strategy

As discussed earlier the overall development strategy is to retrieve from a library a standard subspace generator for the output domain of SWD (sequences over a finite domain), then to specialize it so that only feasible solutions (schedules) are generated, and then to further specialize it so that only optimal solutions are generated. Interleaved with this specialization activity are code optimization steps. We conclude with some consideration of the design and representation of data structures in the generator.

3.3 Algorithm Design

The first step is to select a standard subspace generator that can enumerate a superset of the feasible space SCHEDULES(*Jobs*, *Precedes*). We select a generator $\mathcal{S}(D)$ for sequences over type D that works as follows (see also Figure 1): Each subspace is described by a sequence (called *ps* here, an abbreviation of "partial schedule") denoting the maximal common prefix of sequences in the subspace. Clearly the empty sequence describes the whole type $\mathbf{seq}(D)$ and thus it is the descriptor for the root

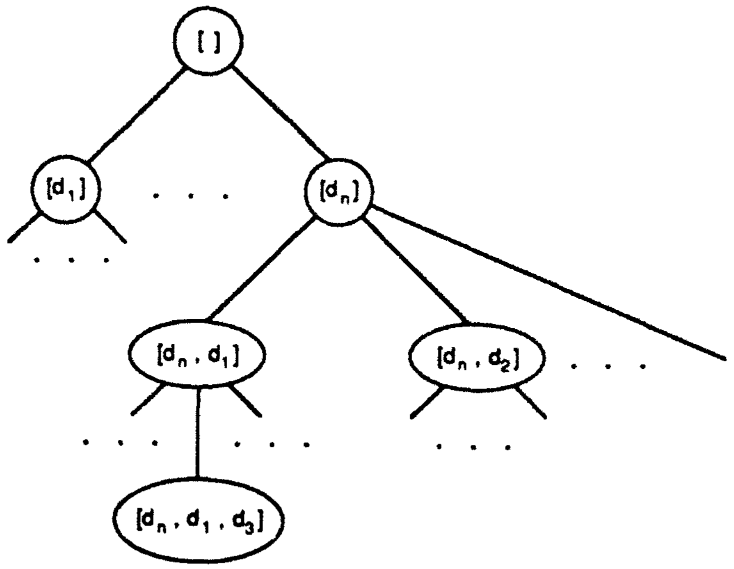

Figure 1: Subspace generator tree for sequences over $\{d_1, d_2, ..., d_n\}$

node (of the subspace tree). A descriptor *ps* is split into descriptors of subspaces by appending a single element. For example, if we want to enumerate sequences over $\{0,1,2\}$ then the subspace descriptor $[2,1]$, which denotes all sequences beginning with a 2 followed by a 1, is split into the descriptors $[2,1,0]$, $[2,1,1]$, and $[2,1,2]$.

This generator is specialized by incorporating some of the constraints of SCHEDULES into the subspace splitting operation. This is done by developing a test that can determine that a subspace descriptor cannot describe any feasible solutions. More precisely we derive a necessary condition of

$$\begin{aligned}\exists\, S\ [&S\ \textit{extends}\ ps\\ &\wedge\ \textit{elements-of}(S) = \textit{Jobs}\\ &\wedge\ \forall j_1, j_2\ \in \textit{Jobs}[\textit{Precedes}(j_1, j_2)\ \Longrightarrow\ \textit{index}(S, j_1)\ <\ \textit{index}(S, j_2)]]\end{aligned}$$

over the variables $\{Jobs, Precedes, Deadline, ps\}$. In words, we derive a necessary condition on the existence of an extension S to subspace descriptor *ps* that is feasible, i.e., an extension that has all the jobs of *Jobs* and no ordering constraint is violated. A derivation such as that in Figure 2 results in the filter $\Phi(ps)$:

$$\begin{tikzcd}[column sep=small]
S \textit{ extends } ps \iff S = concat(ps, qs) \arrow[rrd] & & \\
\textit{elements-of}(S) = Jobs \arrow[rr] & & \textit{elements-of}(concat(ps, qs)) = Jobs \arrow[d, Leftrightarrow] \\
& & \textit{elements-of}(ps) \uplus \textit{elements-of}(qs) = Jobs \arrow[dl, Rightarrow] \arrow[d, Leftrightarrow] \\
& \textit{elements-of}(ps) \subseteq Jobs & \textit{elements-of}(qs) = Jobs - \textit{elements-of}(ps)
\end{tikzcd}$$

$$\forall j_1, j_2 \in Jobs\ [\ Precedes(j_1, j_2) \implies index(S, j_1) < index(S, j_2)]$$

$$\Big\Downarrow\ \textit{case analysis}$$

$$\forall j_1, j_2\ [\ j_1 \in \textit{elements-of}(ps) \wedge j_2 \in \textit{elements-of}(ps)\ \wedge\ Precedes(j_1, j_2)$$
$$\implies index(S, j_1) < index(S, j_2)]$$

$$\&$$

$$\forall j_1, j_2\ [\ j_1 \in Jobs - \textit{elements-of}(ps) \wedge j_2 \in \textit{elements-of}(ps) \implies \neg Precedes(j_1, j_2)]$$

Figure 2: Deriving a filter via forward inference

$$\textit{elements-of}(ps) \subseteq Jobs\ \wedge$$

$$\forall j_1, j_2\ [\ j_1 \in \textit{elements-of}(ps) \wedge j_2 \in \textit{elements-of}(ps) \wedge Precedes(j_1, j_2)$$
$$\implies index(S, j_1) < index(S, j_2)]\ \wedge$$

$$\forall j_1, j_2\ [\ j_1 \in Jobs - \textit{elements-of}(ps) \wedge j_2 \in \textit{elements-of}(ps) \implies \neg Precedes(j_1, j_2)]$$

With this filter we have the first version of our target algorithm, shown in Figure 3. This gives us the rough structure of the target algorithm. Before proceeding there is a simple but powerful simplification that we can perform. Notice that the filter $\Phi(ps)$ holds invariantly on entry to the recursive function SCHEDULES1_aux(ps), yet the straightforward implementation of SCHEDULES1_aux(ps) would test $\Phi(append(ps, a))$. We set up the directed inference task of simplifying $\Phi(append(ps, a))$ under the assumption of $\Phi(ps)$ (see Figure 4) with the result $\phi(ps, a)$:

$$\forall j \in Jobs - \textit{elements-of}(ps)\ [\neg Precedes(j, a)] \wedge a \in Jobs - \textit{elements-of}(ps).$$

```
SCHEDULES1($Jobs, Precedes, Deadline$)=
    if $\Phi([\,])$ then SCHEDULES1_aux([ ])

SCHEDULES1_aux($ps$)=
    assert $\Phi(ps)$ in
    if $size(ps) = size(Jobs)$
        then $\{ps\}$
        else $\bigcup_{a \in Jobs \wedge \Phi(append(ps,a))}$ SCHEDULES1_aux($append(ps,a)$)
```

Figure 3: Subspace Generator for SCHEDULES - Initial Version

Inference-direction	$\Longleftrightarrow$ (logical equivalence)
Assume	$\Phi(ps) \wedge a \in Jobs$
Source	$\Phi(append(ps,a))$

1. $elements\text{-}of(append(ps,a)) \subseteq Jobs$

   $\Longleftrightarrow \; (elements\text{-}of(ps) + a) \subseteq Jobs$

   $\Longleftrightarrow \; a \in (Jobs - elements\text{-}of(ps))$

2. case analysis results in **true**

3. case analysis results in $\forall j \in Jobs - elements\text{-}of(ps)\ [\neg Precedes(j,a)]$

Figure 4: Simplification of $\Phi(append(ps,a))$

```
SCHEDULES2($Jobs, Precedes, Deadline$)=
    SCHEDULES2_aux([ ])

SCHEDULES2_aux($ps$)=
    assert $\Phi(ps)$ in
    if $size(ps) = size(Jobs)$
        then $\{ps\}$
        else $\bigcup_{\substack{a \in Jobs - elements\text{-}of(ps) \\ \forall j \in Jobs - elements\text{-}of(ps)\ [\neg Precedes(j,a)]}}$ SCHEDULES2_aux($append(ps, a)$)
```

Figure 5: Simplified Generator for SCHEDULES

Using $\phi(ps, a)$ in place of $\Phi(append(ps, a))$ results in a large savings of time per recursive call. This optimization is related to finite differencing since we are incrementally maintaining an invariant relation Φ, but different in that the invariant does not involve initializing and updating a data structure. The simplified generator appears in Figure 5.

3.4 Finite Differencing Optimization

We now apply finite differencing in order to further refine and optimize SCHEDULES2. Note that the loop in the recursion involves enumerating over the set

$$\{a \mid a \in Jobs - elements\text{-}of(ps) \land \forall j \in Jobs - elements\text{-}of(ps)[\neg Precedes(j,a)]\}.$$

We wish to reduce the complexity of computing this set by creating some intermediate data structures and maintaining their values incrementally. We can pull out from this expression the following invariants to maintain:

$$\begin{aligned} Free\_set &= Jobs - elements\text{-}of(ps) \\ Test &= \lambda a. \forall j \in Free\_set(ps)[\neg Precedes(j,a)] \\ Min\_set &= \{a \mid a \in Free\_set \land Test(a)\} \end{aligned}$$

As mentioned in Section 2.3, the *Test* expression may be maintained by reformulating it in an equivalent, incrementally computable form. In this case we replace *Test* by

$$size(\{j \mid j \in Free\_set(ps) \land Precedes(j,a)\}) = 0.$$

```
SCHEDULES3($Jobs, Precedes, Deadline$)=
    SCHEDULES3_aux([ ], $ps\_Size\_init$, $Free\_set\_init$,
                        $Prec\_map\_init$, $Size\_map\_init$, $Min\_set\_init$)

SCHEDULES3_aux($ps, ps\_size, Free\_set, Prec\_map, Size\_map, Min\_set$)=
    assert $\Phi(ps) \wedge ps\_size = size(ps)$
      $\wedge\ Free\_set = Jobs - elements\text{-}of(ps)$
      $\wedge\ Prec\_map = \lambda a.\{j \mid j \in Free\_set \wedge Precedes(j, a)\}$
      $\wedge\ Size\_map = \lambda a.size(Prec\_map(a))$
      $\wedge\ Min\_set = \{a \mid a \in Free\_set \wedge Size\_map(a) = 0\}$ in
    if $ps\_size = size(Jobs)$
      then $\{ps\}$
      else reduce($\cup$, {SCHEDULES3_aux($append(ps, a)$, $\Delta\_ps\_size$,
                                          $\Delta\_Free\_set$, $\Delta Prec\_map$,
                                          $\Delta Size\_map$, $\Delta Min\_set$)
                                        $\mid a \in Min\_set$})
```

Figure 6: SCHEDULES plus Finite Differencing Structure

The resulting set of data invariants to maintain, called $\xi(ps)$, is

$$
\begin{aligned}
ps\_size &= size(ps)\\
Free\_set &= Jobs - elements\text{-}of(ps)\\
Prec\_map &= \lambda a.\{j \mid j \in Free\_set \wedge Precedes(j, a)\}\\
Size\_map &= \lambda a.size(Prec\_map(a))\\
Min\_set &= \{a \mid a \in Free\_set \wedge Size\_map(a) = 0\}
\end{aligned}
$$

The challenge is to maintain these invariants under change to the parameter *ps* in SCHEDULES2_aux (they also depend on *Jobs* and *Precedes* but these are fixed parameters and so we treat them as constants). In a recursive setting we must add each of the above variables as new parameters to the recursive function. In effect they become part of the local state of the recursive computations. Figure 6 shows schematically the structural changes induced by our attempt to maintain the invariants $\xi(ps)$. This figure is schematic in that it only indicates where initialization and update code is positioned and not how it is achieved.

The directed inference involved in deriving the initialization and update code for the invariant $ps\_size = size(ps)$ is simple. When *ps* is initialized to [] (the empty sequence), then the value of *ps_size* is 0. To derive the update code we perform equality-preserving inference on the source term $size(append(ps, a))$ under the assumption

$ps\_size = size(ps)$, resulting in the derived term $ps\_size + 1$. Analogous derivations for the initialization and update code for the invariant

$$Size\_map = \lambda b.size(Prec\_map(b))$$

are given in Figure 7.

Incorporating all of the initialization and update code results in the refinement of SCHEDULES3 to SCHEDULES4 in Figure 8. Note however that in SCHEDULES4 we can eliminate a number of dead, or unused, variables and their corresponding update code. The result is SCHEDULES5 in Figure 9, an efficient generator of feasible objects.

3.5 Incorporating the Cost Function

We now further specialize the generator so that it only enumerates optimal cost objects, as described in Section 2.1. Generally the technique is analogous to that used for constraining the generator to enumerate only feasible solutions: we derived a necessary condition on the existence of feasible objects in a subspace. We derive a necessary condition on the existence of optimal solutions. In [34] we show that several common pruning techniques such as lower bound pruning and dominance relations can be derived in this way. Here we focus on the derivation of a lower bound function for SWD and how it can be used to improve the search process of SCHEDULES. In particular we derive a lower bound on the cost function

$$size(\{j \mid j \in Jobs \wedge Deadline(j) < index(S, j)\}).$$

The derivation in Figure 10 results in a bound that is the sum of (1) the number of jobs in the partial schedule ps that have already missed their deadlines and (2) the number of jobs not in ps that have already missed their deadlines (i.e. their deadline lies between 1 and $size(ps)$). Another term which we have derived manually (but not presented in the figure) would measure the number of jobs j not yet in ps that could not possibly meet their deadline (because too many predecessors must be executed before j's earliest possible execution time). This additional term would improve the bound and thus improve performance of the search procedure. The lower bound function can be maintained incrementally just as we've done for $\Phi(ps)$ and the finite differencing variables. The initialization and update code are derived in Figure 11.

Exploiting the lower bound function gives us a classic branch-and-bound algorithm. It records the best schedule found so far in the search process and its cost ub, and deletes from consideration any subspace whose lower bound is not less than ub. To use the lower bound function incrementally we must again add a new parameter to the parameter list of the recursive function in SCHEDULES. In addition we convert to iterative form so that we can more easily maintain global variables that record the least cost solution found so far and its cost (see Figure 12).

$Size\_map = \lambda b.size(\{j \mid j \in Jobs - elements\text{-}of(ps) \wedge Precedes(j,b)\})$

<u>Initialization</u> ps $\leftarrow$ [ ]

Assume
Source $\lambda b.size(\{j \mid j \in Jobs - elements\text{-}of([\,]) \wedge Precedes(j,b)\})$

$= \lambda b.size(\{j \mid j \in Jobs \wedge Precedes(j,b)\})$

<u>Increment</u> ps $\leftarrow$ $append(ps, a)$

```
Assume $Prec\_map = \lambda b.\{j \mid j \in Free\_set \wedge Precedes(j,b)\}$,
        $\Delta Prec\_map = \lambda b.($if $Precedes(a,b)$
                                    then $Prec\_map(b) - a$
                                    else $Prec\_map(b))$,
        $Size\_map = \lambda b.size(Prec\_map(b))$
Source $\lambda b.size\ (\Delta Prec\_map)$

        $= \lambda b.size($if $Precedes(a,b)$
                    then $Prec\_map(b) - a$
                    else $Prec\_map(b))$

        $= \lambda b.($if $Precedes(a,b)$
                then $size(Prec\_map(b) - a)$
                else $size(Prec\_map(b)))$

        $= \lambda b.($if $Precedes(a,b)$
                then $Size\_map(b) - 1$
                else $Size\_map(b))$
```

Figure 7: Incremental Maintenance of *Size_map*

```
SCHEDULES4($Jobs$, $Precedes$, $Deadline$)=
    let $ps\_size = 0$
        let $Free\_set = Jobs$
            let $Prec\_map = \lambda b.\{j \mid j \in free\_set \wedge Precedes(j,b)\}$
                let $Size\_map = \lambda b.size(Prec\_map(b))$
                    let $Min\_set = \{a \mid a \in free\_set \wedge Size\_map(a) = 0\}$
                        SCHEDULES4_aux([ ], $ps\_size$, $Free\_set$,
                                                $Prec\_map$, $Size\_map$, $Min\_set$)

SCHEDULES4_aux($ps, ps\_size, Free\_set, Prec\_map, Size\_map, Min\_set$)=
    assert $\Phi(ps) \wedge \xi(ps)$ in
    if $ps\_size = size(Jobs)$
        then $\{ps\}$
        else reduce ($\cup$, {SCHEDULES4_aux($append(ps,a)$,
                                    $ps\_size + 1$,
                                    $free\_set - a$,
                                    $\lambda$ $b$. (if $Precedes(a,b)$
                                            then $Prec\_map(b) - a$
                                            else $Prec\_map(b)$)),
                                    $\lambda$ $b$.(if $Precedes(a,b)$
                                            then $Size\_map(b) - 1$
                                            else $Size\_map(b)$)),
                                     $(Min\_set \cup \{b \mid Precedes(a,b) \wedge Size\_map(b) = 1\}) - a)$
                               $\mid a \in Min\_set\})$
```

Figure 8: SCHEDULES with Finite Differencing

```
SCHEDULES5($Jobs$, $Precedes$, $Deadline$)=
    let $ps\_size = 0$
            let $Size\_map = \lambda b.size(\{j \mid j \in Jobs \ \wedge \ Precedes(j,b)\})$
                let $Min\_set = \{a \mid a \in free\_set \ \wedge \ Size\_map(a) = 0\}$
                    SCHEDULES5_aux([ ], $ps\_size$, $Size\_map$, $Min\_set$)

SCHEDULES5_aux($ps, ps\_size, Size\_map, Min\_set$)=
assert $\Phi(ps) \ \wedge \ \xi(ps)$ in
if $ps\_size = size(Jobs)$
    then $\{ps\}$
    else reduce($\cup$, {SCHEDULES5_aux($append(ps,a), ps\_size + 1$,
                                              $\lambda$ $b$. (if $Precedes(a,b)$
                                                      then $Size\_map(b) - 1$
                                                      else $Size\_map(b)$),
                                              $(Min\_set \cup \{b \mid Precedes(a,b) \wedge Size\_map(b) = 1\}) - a)$
                                      $\mid a \in Min\_set\}$)
```

Figure 9: SCHEDULES with dead variables removed

Another reason for converting to iterative form is to make the recursive calls into explicit data structures; this allows us to more flexibly specify control strategies. The data structure *PQ* is specified as an abstract data type known as a priority queue (or agenda). Generally priority queues represent sets of objects with associated priorities. The operations include *Init_PQ* which creates an empty queue, *Insert* which adds an object to the queue and assigns it a priority, and *Select* which extracts that object in the queue with highest priority. Priority queues can be used for controlling search processes by letting the objects be representations of tasks and letting the priority of a task reflect the desired search strategy. For example, to achieve a best-first search, the priority on a task should be the value of the lower bound function. Breadth-first and depth-first search can be achieved by basing the priority of a subspace on its depth in the search tree. At this point we leave the specification of the priority open.

3.6 Data Structure Representation

The next step in the design process is to develop representations for each of the abstract data types in the algorithm. The representation decisions for a variable depend on the operations performed on it and their frequency of occurence. Consider the subspace descriptor *ps* which denotes a partial schedule:

Inference-direction	$\geq$
Target-characteristics	$\mathit{varset} \subseteq \{\mathit{Jobs}, \mathit{Precedes}, \mathit{Deadline}, \mathit{ps}\}$
Assume	$S \text{ extends } \mathit{ps}$
	$\wedge\ S = \mathit{concat}(\mathit{ps}, \mathit{qs})$
	$\wedge\ \mathit{elements\text{-}of}(s) = \mathit{Jobs}$
	$\wedge\ \Phi(\mathit{ps})\ \wedge \xi(\mathit{ps})$
Source	$\mathit{size}(\{j \mid j \in \mathit{Jobs} \wedge \mathit{Deadline}(j) < \mathit{index}(S,j)\})$

$$\begin{aligned}
&\mathit{size}(\{j \mid j \in \mathit{Jobs} \wedge \mathit{Deadline}(j) < \mathit{index}(S,j)\})\\
&=\mathit{size}(\{j \mid j \in \mathit{elements\text{-}of}(S) \wedge \mathit{Deadline}(j) < \mathit{index}(S,j)\})\\
&=\mathit{size}(\{j \mid j \in \mathit{elements\text{-}of}(\mathit{concat}(\mathit{ps},\mathit{qs})) \wedge \mathit{Deadline}(j) < \mathit{index}(S,j)\})\\
&=\mathit{size}(\{j \mid (j \in \mathit{elements\text{-}of}(\mathit{ps}) \vee j \in \mathit{elements\text{-}of}(\mathit{qs}))\ \wedge\ \mathit{Deadline}(j) < \mathit{index}(S,j)\})\\
&=\mathit{size}(\{j \mid j \in \mathit{elements\text{-}of}(\mathit{ps}) \wedge \mathit{Deadline}(j) < \mathit{index}(S,j)\}\\
&\qquad \cup \{j \mid j \in \mathit{elements\text{-}of}(\mathit{qs}) \wedge \mathit{Deadline}(j) < \mathit{index}(S,j)\}\})\\
&=\mathit{size}(\{j \mid j \in \mathit{elements\text{-}of}(\mathit{ps}) \wedge \mathit{Deadline}(j) < \mathit{index}(S,j)\}\})\\
&\qquad +\mathit{size}(\{j \mid j \in \mathit{Jobs} - \mathit{elements\text{-}of}(\mathit{ps}) \wedge \mathit{Deadline}(j) < \mathit{index}(S,j)\}\}))\\
&\geq \mathit{size}(\{j \mid j \in \mathit{elements\text{-}of}(\mathit{ps}) \wedge \mathit{Deadline}(j) < \mathit{index}(S,j)\}\})\\
&\qquad +\mathit{size}(\{j \mid j \in \mathit{Jobs} - \mathit{elements\text{-}of}(\mathit{ps}) \wedge \mathit{Deadline}(j) \leq \mathit{size}(\mathit{ps})\}\}))
\end{aligned}$$

Figure 10: Deriving the Lower Bound Function

$$lb(ps) = size(\{j \mid j \in elements\text{-}of(ps) \wedge Deadline(j) < index(S,j)\})$$

Initialize: $ps \leftarrow [\,]$

Source $lb([\,])$
$=size(\{j \mid j \in elements\text{-}of([\,]) \wedge \ldots\})$
$=0$

Increment: $ps \leftarrow append(ps,a)$

Inference-direction =
Assume $lb(ps) = size(\{j \mid j \in elements\text{-}of(ps) \wedge Deadline(j) < index(ps,j)\})$
Source $size(\{j \mid j \in elements\text{-}of(append(ps,a)) \wedge Deadline(j) < index(append(ps,a),j)\})$

$size(\{j \mid j \in elements\text{-}of(append(ps,a)) \wedge Deadline(j) < index(append(ps,a),j)\})$

$=size(\{j \mid j \in elements\text{-}of(ps) \wedge Deadline(j) < index(ps,j)\})$
$+size(\{j \mid j = a \wedge Deadline(j) < index(append(ps,a),j)\})$

$=lb(ps) + (\textbf{if } Deadline(a) < size(ps) + 1 \textbf{ then } 1 \textbf{ else } 0)$

Figure 11: Incremental maintenance of the lower bound function

```
SWD6($Jobs, Precedes, Deadline$)=
let     $solution$ : seq($JOB$) = [ ],
        $lb$ : natno = 0,
        $ub$ : natno = $maxint$,
        $Job\_size$ : natno = $size(Jobs)$,
        $ps$ : seq($Jobs$) = [ ],
        $ps\_size$ : natno = 0,
        $Size\_map$ : ($Jobs \rightarrow$ natno) = $\lambda b.\ size(\{j \mid j \in Jobs \wedge Precedes(j,b)\})$
let     $Min\_set$ : set($JOB$) = $\{a \mid a \in Jobs \wedge Size\_map(a) = 0\}$,
        $PQ$ : $Priority\_Queue$(seq($JOB$) × natno × ($Jobs \rightarrow$ natno) × set($JOB$) × natno)
              = $Init\_PQ$;
$Insert(PQ, \langle ps, ps\_size, Size\_map, Min\_set, lb\rangle)$;
while $nonempty(PQ)$ do
        begin
        $\langle ps, ps\_size, Size\_map, Min\_set, lb\rangle \leftarrow Select(PQ)$;
        if $lb < ub$ then
            if $ps\_size = Job\_size$
                then % record best solution and cost found so far
                  $solution \leftarrow ps$; $ub \leftarrow lb$;
                else enumerate $a \in Min\_set$ do
                    $Insert(PQ, \langle append(ps, a)$,
                        $ps\_size + 1$,
                        $\lambda b.$(if $Precedes(a,b)$
                            then $Size\_map(b) - 1$
                            else $Size\_map(b)$),
                        $(Min\_set \cup \{b \mid Precedes(a,b) \wedge Size\_map(b) = 1\}) - a$,
                        $lb$ + (if $Deadline(a) \leq ps\_size$ then 1 else 0)$\rangle$)
        end;
 $solution$
end.
```

Figure 12: SWD - Iterative Optimization Version

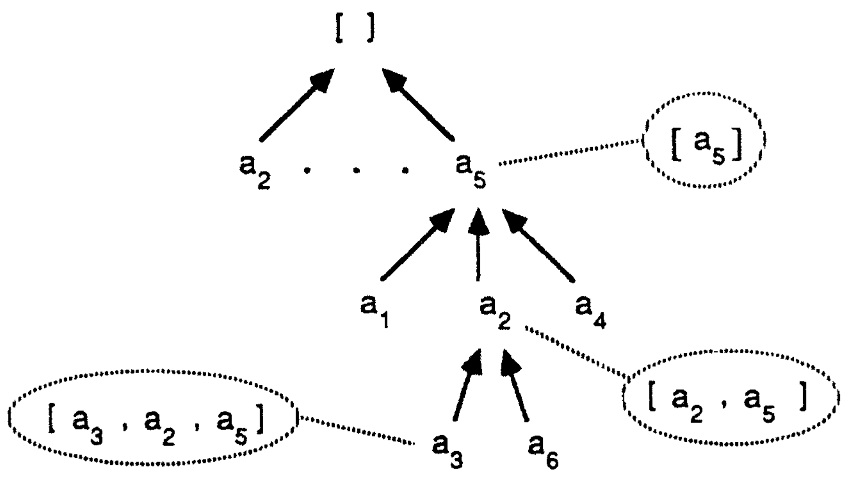

Figure 13: A Structure-Sharing Representation of Sequences

Operation	*Frequency*
$ps \leftarrow [\,]$	once
$append(ps, a)$	often
$solution \leftarrow ps$	occasionally

A standard representation for sequences is linked lists; however, this representation is expensive for *ps* because it entails copying *ps* every time the *append* operation is performed. A better representation is shown in Figure 13 where alternative versions of *ps* coexist and share common structure. The data structure *ps* is simply a pointer to the last element of the sequence. In this representation, initialization and *append* take constant time, and the assignment operation takes time linear in the size of *ps* (by tracing upwards from the element pointed to by *ps*).

This representation is the composition of two data type conversions: that a sequence may be represented as the reverse of another sequence, and that a sequence may be represented as a linked list. The facts that motivate this representation are that (1) prepending an element onto a sequence does not require copying the sequence (provided that random modifications to the sequence are not performed), and (2) appending an element onto a sequence is equivalent to prepending the element onto the reverse of the sequence (provided that the reversed sequence is reversed again when the value of it is desired). An analogous structure-sharing representation can be used on the mapping variable *Size_map*.

On the other hand, if a depth-first strategy is chosen, then *ps* is accessed in a stack manner, with no pointers to subsequences, so can be implemented as an array with a top pointer. We thus see that the context of use of a data object *affects the choice* of a representation for the data object.

4 Concluding Remarks

We have formally developed specification SWD into algorithm SWD6. The initial specification was a concise and straightforward definition of the problem: optimal scheduling of jobs on a processor under precedence constraints and deadlines. It has essentially no algorithmic content. The derived algorithm is much longer and more difficult to understand, yet is guaranteed to be correct with respect to the specification.

The development process produced a series of increasingly efficient versions of the algorithm. The initial specification SWD was uncompilable using conventional techniques. The initial subspace generator searched a tree with $O(n^n)$ nodes and spent $O(n^4)$ time per node (where n denotes the number of jobs to be scheduled). Introducing the filter Φ and simplifying it reduced the number of nodes searched to between $O(n)$ and $O(n!)$ depending on the strength of the precedence relation *Precedes*, and reduced the time spent per node to $O(n^2)$. Finite differencing optimizations substantially reduced the constant associated with the time spent per node. Introducing the lower bound function and its incremental computation substantially reduced the number of nodes searched although it is difficult to quantify the effect. Finally the introduction of specialized data representations for the data structures of the algorithm further reduced the time spent per node to between $O(n)$ and $O(n \log n)$ depending again on the strength of *Precedes*. Since the nodes can be processed independently further improvements could be made by transforming the algorithm for execution in a parallel environement.

Experimental systems under development at Kestrel can currently support most of the capability needed to perform the derivation of SWD6. The directed inferences in this paper are easily handled by the RAINBOW II inference system. The CYPRESS II algorithm design system has semiautomatically produced subspace generators for several problems, but does not yet support the optimization structure described above (e.g. derivation and use of the lower bound function). The MEQ [5] finite differencing system has produced the initialization and update code described in our example, although in a slightly different format. MEQ allows programmers to specify constraints of the form $c = E$ and it combines analysis, table lookup, and composition to obtain maintenance code for c with respect to modifications to parts of E. Initialization and update code are automatically added to the program during compilation. The DSS system [19] and extensions [10] allow limited transformational development of representations for VHL data structures based on performance criteria. Some of the data structure representations discussed above would require extensions to this system. The REFINE™ language is well-suited for expressing the initial specification SWD. Its compiler can produce executable code from the intermediate forms, although they would be relatively inefficient. The final derived algorithm would be compiled into efficient COMMON LISP code.

We have not yet attempted to integrate all of these systems and develop a uniform interface to the programmer. Some of the substantial difficulties with integration and interface include:

- Explanation - Why was that decision made? What decisions have been made so far? What methods are available for achieving goal G?
- Assistance - What can I do now? What should I do now?
- Assessment - How well am I doing? Which decision is better?
- Instruction - You (the computer) should know x.
- Exploration - What if I do this?

Acknowledgements

We would like to thank Lee Blaine, Cordell Green, Richard Jüllig, Peter Ladkin, and Stephen Westfold for reviewing drafts of this paper. This research was supported in part by the Office of Naval Research (ONR) under Contract N00014-84-C-0473 and in part by the Defense Advanced Research Projects Agency (DARPA) under contract N00014-81-C-0582 monitored by the Office of Naval Research. The views and conclusions contained in this paper are those of the authors and should not be construed as representing the official policies, either expressed or implied, of ONR, DARPA, or the U.S. Government.

References

[1] ABRAIDO-FANDIÑO, L. An overview of *REFINE*^{TM} 2.0. In *Proceedings of the Second International Symposium on Knowledge Engineering* (Madrid, Spain, April 8–10, 1987).

[2] ANDERSON, P. Finite differencing and the specialization system. Forthcoming report, Carnegie-Mellon University, 1987.

[3] BARSTOW, D. R. Domain specific automatic programming. *IEEE Transactions on Software Engineering SE-11*, 11 (November 1985), 1321–1336.

[4] BARSTOW, D. R. *Knowledge-Based Program Construction.* North-Holland, New York, 1979.

[5] COURSEY, J. V. *Knowledge-Based Compilation Using Finite Differencing Techniques.* Tech. Rep. KES.U.85.6, Kestrel Institute, May 1985.

[6] DOWSON, M. ISTAR—an integrated project support environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* (Pittsburgh, PA, April 23–25, 1984), ACM, pp. 132–140. (*ACM SIGPLAN Notices*, 19(5), May 1984; *ACM Software Engineering Notes*, 9(3), May 1984).

[7] FEATHER, M. S. A survey and classification of some program transformation approaches and techniques. In *IFIP TC 2 Working Conference on Program Specification and Transformation*, L. Meertens, Ed., North-Holland, Amsterdam, 1987.

[8] FEATHER, M. S. *A System for Developing Programs by Transformation.* PhD thesis, University of Edinburgh, 1979.

[9] GOLDBERG, A., AND KOTIK, G. Knowledge-based programming: an overview of data and control structure refinement. In *Software Validation: Inspection, Testing, Verification, Alternatives*, H. Hausen, Ed., Elsevier, 1984, pp. 287–309. Technical Report KES.U.83.7, Kestrel Institute, October 1983.

[10] GOLDBERG, A., AND SMITH, D. R. *Towards a Performance Estimation Assistant.* Tech. Rep. KES.U.86.10, Kestrel Institute, November 1986.

[11] GOLDBERG, A. T. Knowledge-based programming: a survey of program design and construction techniques. *IEEE Transactions on Software Engineering SE-12*, 7 (July 1986), 752–768. Technical Report KES.U.86.1, Kestrel Institute, July 1986.

[12] GOPINATH, K., AND HENNESSY, J. *Copy Elimination with Abstract Interpretation.* Tech. Rep. KLASSIC-87-17, Stanford University Computer Science Department, February 1987.

[13] GREEN, C., LUCKHAM, D., BALZER, R., CHEATHAM, T., AND RICH, C. *Report on a Knowledge-Based Software Assistant.* Tech. Rep. KES.U.83.2, Kestrel Institute, July 1983.

[14] GREEN, C., AND WESTFOLD, S. Knowledge-based programming self-applied. In *Machine Intelligence 10*, J. E. Hayes, D. Mitchie, and Y. Pao, Eds., Wiley, New York, 1982, ch. 17, pp. 339-359. Technical Report KES.U.84.1, Kestrel Institute, February 1984.

[15] JÜLLIG, R., POLAK, W., LADKIN, P., AND GILHAM, L. *Knowledge-Based Software Project Management* Tech. Rep. KES.U.87.3, Kestrel Institute, November 1986.

[16] KANT, E. On the efficient synthesis of efficient programs. *Artificial Intelligence 20*, 3 (May 1983), 253-306.

[17] KEDZIERSKI, B. I. Knowledge-based project management and communication support in a system development environment. In *Proceedings of the 4th Jerusalem Conference on Information Technology* (Jerusalem, May 20-25, 1984). Technical Report KES.U.84.2, Kestrel Institute, April 1984.

[18] KORF, R. E. Toward a model of representation changes. *Artificial Intelligence 14*, 1 (August 1980), 41-78.

[19] KOTIK, G. *Knowledge-based Compilation of High-Level Data Types.* Tech. Rep. KES.U.83.5, Kestrel Institute, March 1983.

[20] KRUTCHEN, P., SCHONBERG, E., AND SCHWARTZ, J. Software prototyping using the SETL programming language. *IEEE Software 1*, 4 (October 1984), 66-76.

[21] MÖLLER, B., AND PARTSCH, H. Formal specification of large-scale software—objectives, design decisions and experiences in a concrete software project. In *IFIP TC 2 Working Conference on Program Specification and Transformation*, L. Meertens, Ed., North-Holland, Amsterdam, 1987, pp. 491-516.

[22] PAIGE, R., AND KOENIG, S. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems 4*, 3 (July 1982), 402-454.

[23] PARTSCH, H. Transformational program development in a particular problem domain. *Science of Computer Programming 7*, 2 (September 1986), 99-242.

[24] PARTSCH, H., AND STEINBRÜGGEN, R. Program transformation systems. *ACM Computing Surveys 15*, 3 (September 1983), 199-236.

[25] PENEDO, M. H., AND STUCKLE, E. D. PMDB - a project master database for software engineering environments. In *8th International Conference on Software Engineering* (London, UK, August 28-30, 1985), IEEE, pp. 150-157.

[26] PERRY, D. Version control in the Inscape environment. In *9th International Conference on Software Engineering* (Monterey, CA, March 30–April 2, 1987), pp. 142–149.

[27] PHILLIPS, J. *Self-Described Programming Environments.* PhD thesis, Stanford University Computer Science Department, March 1983. Technical Report KES.U.83.1, Kestrel Institute, March 1983.

[28] POLAK, W. Framework for a knowledge-based programming environment. In *Advanced Programming Environments (Proceedings of an International Workshop)*, R. Conradi, T. M. Didriksen, and D. H. Wanvik, Eds., Springer-Verlag, Berlin, 1987, pp. 566–576. Technical Report KES.U.86.3, Kestrel Institute, June 1986.

[29] SCHERLIS, W. Program improvement by internal specialization. In *Eighth ACM Symposium on Principles of Programming Languages* (Williamsburg, VA, January 1981), ACM, pp. 41–49.

[30] SCHERLIS, W. L. Abstract data types, specialization, and program reuse. In *Advanced Programming Environments*, R. Conradi, Ed., Springer-Verlag, Berlin, 1987, pp. 433–453. Lecture Notes in Computer Science, Vol. 244.

[31] SCHONBERG, E., SCHWARTZ, J., AND SHARIR, M. An automatic technique for the selection of data representations in SETL programs. *ACM Transactions on Programming Languages and Systems 3*, 2 (April 1981), 126–143.

[32] SMITH, D. R. Derived preconditions and their use in program synthesis. In *Sixth Conference on Automated Deduction* (Berlin, 1982), D. W. Loveland, Ed., Springer-Verlag, pp. 172–193. Lecture Notes in Computer Science, Vol. 138.

[33] SMITH, D. R. On the design of generate-and-test algorithms: subspace generators. In *Program Specification and Transformation*, L. Meertens, Ed., North-Holland, Amsterdam, 1987, pp. 207–220.

[34] SMITH, D. R. Subspace generators and the design of optimization algorithms. In preparation, 1987.

[35] SMITH, D. R. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence 27*, 1 (February 1985), 43–96. Reprinted in *Readings in Artificial Intelligence and Software Engineering*, C. Rich and R. Waters, Eds., Los Altos, CA, Morgan Kaufmann, 1986.

[36] SMITH, D. R., KOTIK, G. B., AND WESTFOLD, S. J. Research on knowledge-based software environments at Kestrel Institute. *IEEE Transactions on Software Engineering SE-11*, 11 (November 1985), 1278–1295. Technical Report KES.U.85.7, Kestrel Institute, June 1985.

[37] TRAUGOTT, J. Deductive synthesis of sorting algorithms. In *Eighth Conference on Automated Deduction* (Berlin, 1986), J. H. Siekman, Ed., Springer-Verlag. pp. 394–402. Lecture Notes in Computer Science, Vol. 230.

[38] WESTFOLD, S. Very-high-level programming of knowledge representation schemes. In *Proceedings of the 1984 National Conference on Aritificial Intelligence* (Austin, TX, August 6–10, 1984), AAAI. Technical Report KES.U.84.7, Kestrel Institute, March 1985.

[39] WESTFOLD, S. J., MARKOSIAN, L. Z., AND BREW, W. A. Knowledge-based software development from requirements to code. In *Innovative Software Factories and Ada*, V. Montanari and A. N. Haberman, Eds., Springer-Verlag, Berlin, 1987. To appear.